Besturingssystemen theorie

Academiejaar 2018 – 2019

HO
GENT

Inhoudsopgave

HO
GENT

Hoofdstuk 3: Scheduling

1. Noodzaak tot scheduling
2. Doelstellingen van scheduling
3. Systeembeeld van een proces
4. Strategieën voor low-level scheduling

HO
GENT

1. Noodzaak tot scheduling

→om efficiënt middelen (bronnen, resources) in te zetten om de taken (opdrachten, jobs) uit te voeren.

Bij multiprogrammering (multi-user) moet het OS efficiënt aan de verschillende behoeften van vele gebruikers voldoen. Processen moeten resources benaderen en binnen een redelijke tijd uitgevoerd worden.

HO GENT 5

Bij multiprogrammering[1] moet het besturingssysteem aan de verschillende behoeften van vele gebruikers voldoen. Het moet ook efficiënt zijn. Processen moeten resources benaderen en binnen een redelijke tijd moeten de processen worden uitgevoerd. De kans hierop is groter als de machinecode voor een proces, of tenminste een deel daarvan, in het geheugen staat. Multiprogrammering is de mogelijkheid de code van meerdere processen gelijktijdig in het geheugen te bewaren.
We moeten echter benadrukken dat **multiprogrammering niet hoeft te betekenen dat er veel processen gelijktijdig worden uitgevoerd**. Dit is iets dat vaak verkeerd wordt begrepen. Het geheugen kan van vele processen tegelijk de code bevatten, maar als er maar één CPU is, kan er maar één proces tegelijk worden uitgevoerd. Multiprogrammering is iets anders dan multiprocessing.

[1]multiprogrammering=schakelen tussen verschillende programma's die in het geheugen zijn opgeslagen, wordt ook multi-user genoemd

1. Noodzaak tot scheduling (2)

→ is dus een belangrijk concept in het ontwerp van **multi-user-** en **multiprocessing-besturingssystemen** en in het ontwerp van een **realtimebesturingssysteem**.

Scheduling verwijst dus naar de manier waarop processen prioriteiten worden gegeven. Deze taak wordt uitgevoerd door software die bekend staat als een **scheduler**.

HO GENT 6

Multiprocessing betekent dat het systeem meerdere processors bevat. Elke CPU kan de code van een afzonderlijk proces uitvoeren.
Multi-programmering in een systeem met één CPU levert vele interessante problemen op. De verschillende processen willen allemaal toegang tot de CPU.

De processor (CPU) moet regelmatig lange tijd wachten op in- en uitvoer van een bepaald proces. Tijdens deze wachttijd kan dan een deel van een *ander* proces uitgevoerd worden. Om te bepalen welk proces op welk moment mag uitgevoerd worden, wordt een **scheduler** gebruikt. De scheduler moet de processorbelasting balanceren en voorkomen dat één proces alle CPU-tijd gebruikt, of juist geen CPU-tijd krijgt.
In realtime omgevingen, zoals industriële robots, zorgt de scheduler er ook voor dat processen zich aan hun deadline kunnen houden; dit is cruciaal om het systeem stabiel te houden.

De term **scheduler** wordt ook gebruikt als benaming voor een programma dat op gezette tijden andere programma's start. Een voorbeeld hiervan is het programma **cron** in Unix-achtige besturingssystemen zoals **Linux**. Scheduling met cron gebeurt op een enkele machine. Scheduling op meerdere machines kan met Cronacle van Redwood, AutoSys van ComputerAssociates of Tivoli Workload Scheduler van IBM. Databases zoals Oracle en MySQL kennen ook een ingebouwd schedulingmechanisme.

In dit hoofdstuk bespreken we de scheduling-strategieën van een besturingssysteem, en proberen we eveneens te bepalen hoe goed de verschillende methodes werken.

2. Doelstellingen van scheduling

→doelmatigheid en tevredenheid van de gebruiker

→resources moeten effectief/efficiënt gebruikt worden

→op een snelle en rendabele manier

HO GENT 7

Veel scheduling-strategieën hebben doelmatigheid en tevredenheid van de gebruiker tot doel. Maar ook de resources moeten effectief worden gebruikt. Met andere woorden, niet alleen moet het besturingssysteem iedereen gelukkig maken, maar dat moet ook snel en rendabel gebeuren.

De efficiëntie wordt niet bepaald door één metriek, maar kent meerdere metrieken die in evenwicht wordt gebracht door de scheduler.

We bespreken hierna de verschillende metrieken waarmee een scheduler rekening houdt:

- Doorvoersnelheid (troughput)
- Responstijd
- Consistentie
- Houd de processor aan het werk
- Prioriteiten
- Real-time systemen

2. Doelstellingen van scheduling

- **Doorvoersnelheid (troughput)**
  = aantal processen/tijdseenheid door het systeem
  – Gevolg:
    - Lage doorvoersnelheid -> weinig processen
    - Hoge doorvoersnelheid -> veel processen
      – Lijkt het interessantst
      – Maar geen rekening met procesgrootte
        => meest efficiënte systeem kan sommige processen negeren
        => redelijkheid opgeofferd aan efficiëntie

HO GENT 9

2. Doelstellingen van scheduling

- **Responstijd**
  - Interactieve gebruikers -> snelle respons
  - Batch-gebruikers -> redelijke responstijd
  - Snelle reactie op elk proces
    => meer processen/tijdseenheid
    - Responstijd lijkt hetzelfde als doorvoersnelheid
  - Doorvoersnelheid kan vergroot worden door enkel korte processen te behandelen en lange processen te negeren
    => lange processen worden niet beantwoord

HO GENT 10

2. Doelstellingen van scheduling

- **Consistentie**
  - Eisen gesteld aan systeem:
    zelfde werking om 10u en om 15u
    => responstijd moet ongeveer gelijk zijn
  - Als respons sterk varieert, dan weten de gebruikers niet wat ze mogen verwachten
    => problemen met werkindeling

HO GENT 11

2. Doelstellingen van scheduling

- **Houd de processor aan het werk**
  - BS moet resources aan het werk houden
  - Vb. I/O-processors
    - => meer aandacht aan processen die veel I/O vragen
    - => meer processors aan het werk => meer gedaan
      Als ze allemaal bezig zijn
      => geen I/O request meer genereren
    - => CPU verwerkt andere processen die op CPU wachten
    - => BS moet minder ingrijpen
    - => systeem efficiëntie neemt toe

Ideale geval: BS houdt elke processor aan het werk, zonder deze zwaar te belasten

Afhankelijk van mix van processen in systeem

HO GENT 12

2. Doelstellingen van scheduling

- **Prioriteiten**
  - Elk proces krijgt een prioriteit
    - Hoe hoger hoe belangrijker
  - BS baseert zich hierop bij de scheduling
  - Levert ook problemen:
    - Wie bepaalt de prioriteiten?
    - Wie kent ze toe?
    - Welke richtlijnen zijn er?
    - Wanneer is een proces belangrijker en verdient het daarom een hogere prioriteit?
    - Wat gebeurt er als scheduling op basis van prioriteiten de systeemefficiëntie verlaagt?

2. Doelstellingen van scheduling

- **Real-time systemen**
  - Snelle respons voor besturing van doorgaand proces
    => hoogste prioriteit
  - Rechtvaardigheid en systeemefficiëntie
    => lagere prioriteit voor de andere processen

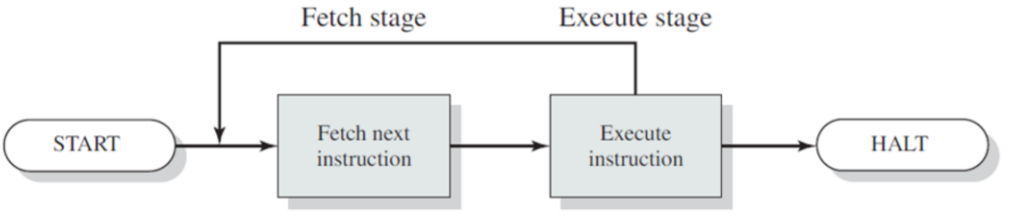
HO GENT

Scheduling is geen eenvoudige zaak: processen eisen soms verschillende zaken, en de taak van de scheduler is om dit voor elk process zo veel mogelijk tegemoet te komen. Wat voor het ene proces efficiënt is, is dit niet voor het andere.
Er wordt door de scheduler een optimalisatie nagestreefd tussen de verschillende metrieken.

Het precieze algoritme van een scheduler is dus behoorlijk complex – dit valt buiten de scope van deze cursus.

3. Uitvoeren van een proces

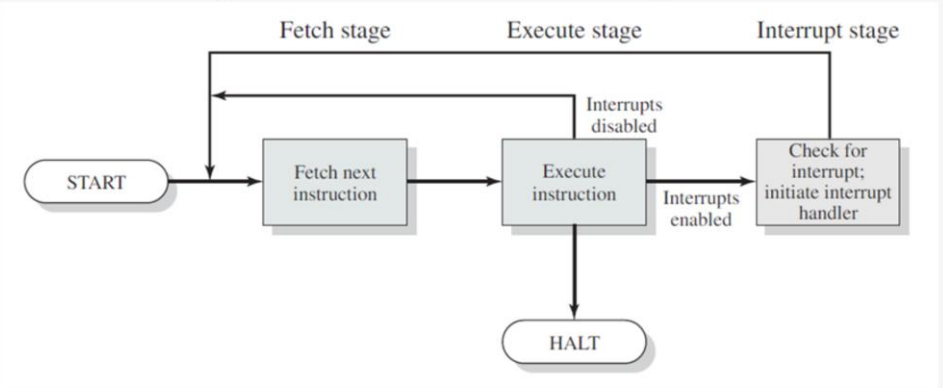
Instructiecyclus zonder onderbrekingen



HO GENT

3. Uitvoeren van een proces

Instructiecyclus met onderbrekingen

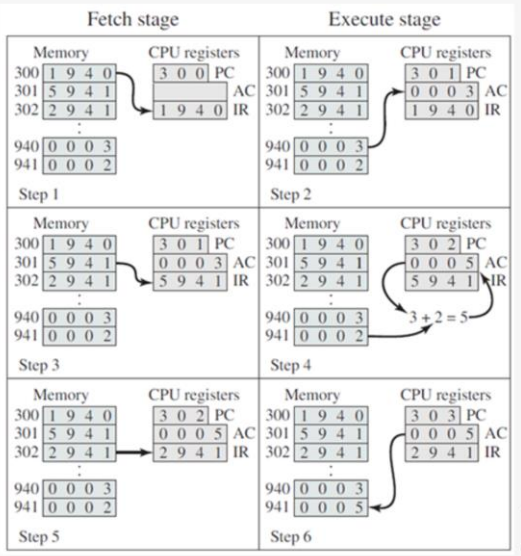


HO GENT

3. Uitvoeren van een proces

PC: Program Counter
AC: Accumulator
IR: Instruction Register

1XXX: copy to AC
2XXX: copy from AC
5XXX: add to AC



HO GENT

4. Systeembeeld van een proces

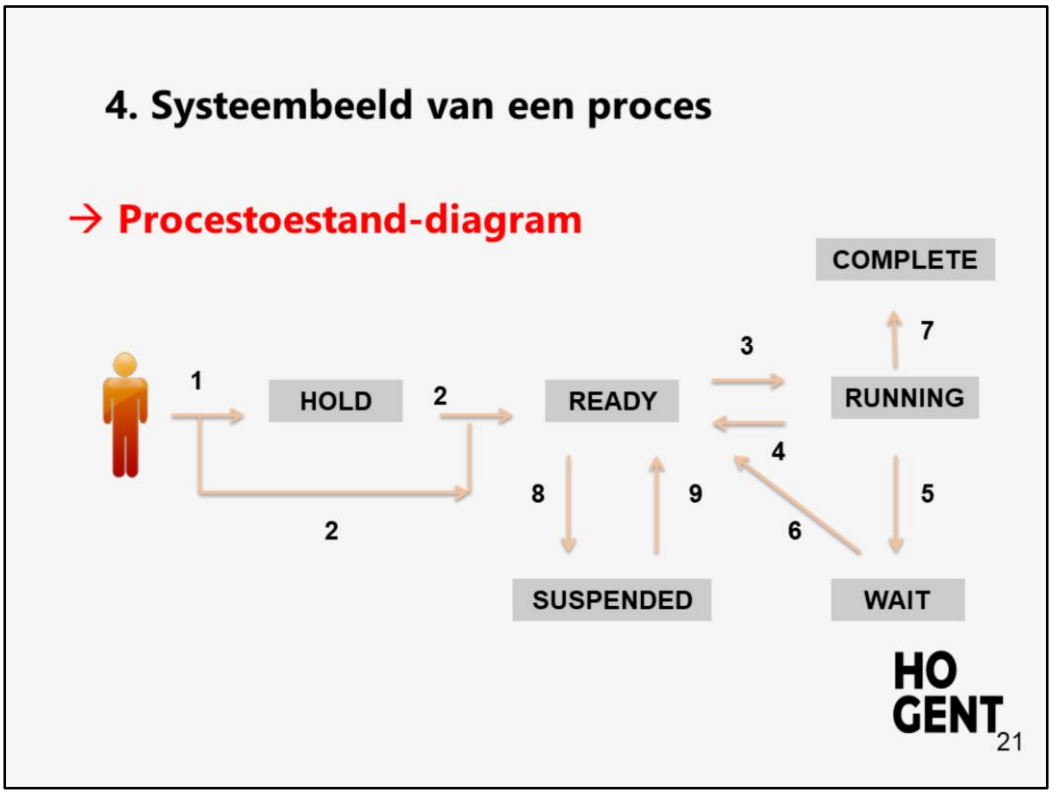
- **Proces bestaat uit**
  - Context
    - ID / nummer
    - Status of toestand
    - PC (Proces Control)
    - Prioriteit
  - Instructies
  - Data

HO GENT 19

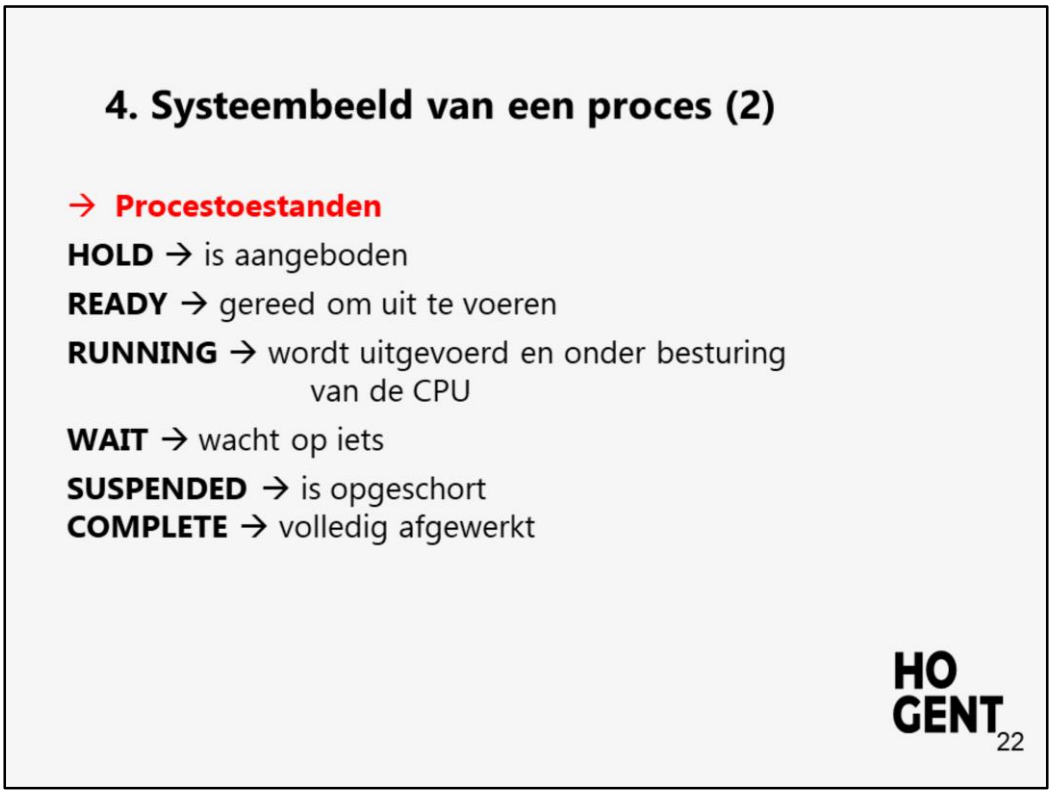
4. Systeembeeld van een proces

- **BS**
  - Bekijkt de processen die het computersysteem binnenkomen
  - **Procestoestand**
    = status
    - Gedefinieerd in termen van de mogelijkheid om te worden uitgevoerd
    - Rechthoek in procestoestandsdiagram (zie volgende slide)
  - **Toestandsovergangen**
    - Pijlen in procestoestandsdiagram (zie volgende slide)

HO GENT 20



Bij scheduling is het nodig de processen zoals deze de handen van de gebruiker verlaten en het computersysteem binnenkomen, zorgvuldig te bekijken. Wij gaan na hoe een besturingssysteem de verschillende processen bekijkt. Laten wij de toestand (status) van een proces definiëren in termen van de mogelijkheid te worden uitgevoerd. Het procestoestand-diagram toont de toestanden waarin een proces kan bestaan. De rechthoeken in het diagram vertegenwoordigen de toestanden. De pijlen ertussen vertegenwoordigen wijzigingen in de toestand: toestandsovergangen.



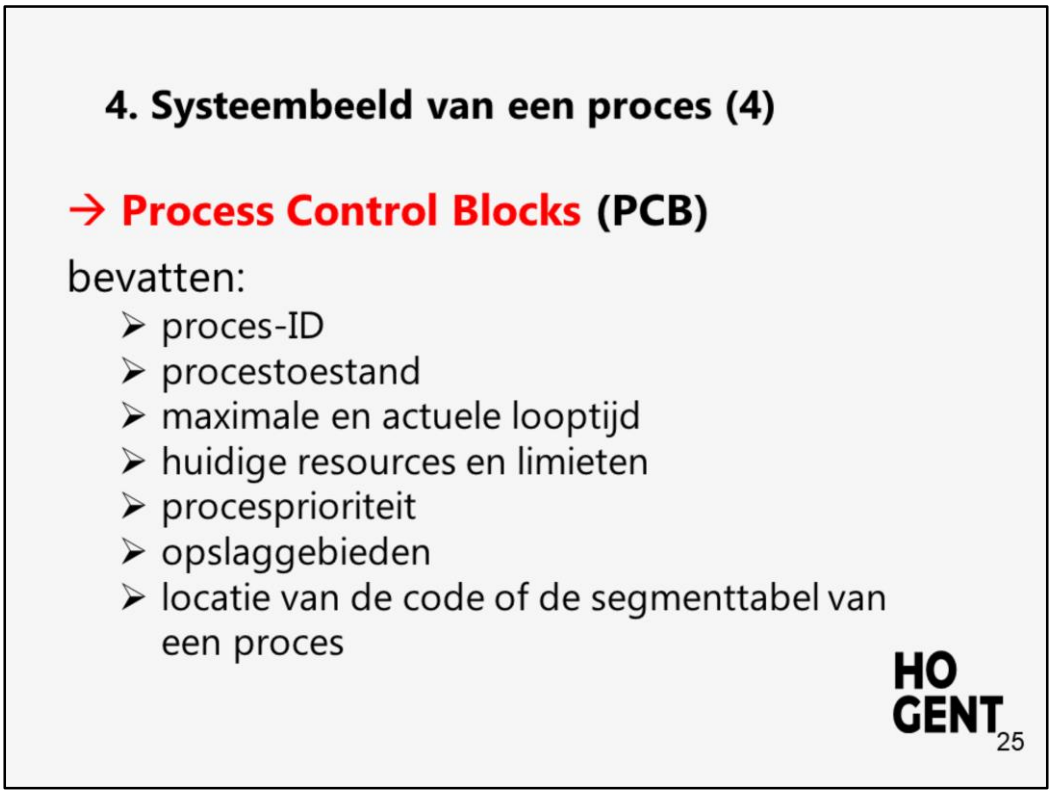
* De toestand **HOLD** duidt aan dat het proces is aangeboden. Het besturingssysteem staat het alleen nog niet toe te worden uitgevoerd of resources aan te vragen. Deze toestand treedt op wanneer iemand een proces aanbiedt met de instructies dat het later moet worden gestart. Veel systemen kunnen een aangeboden proces initialiseren op een bepaald tijdstip, of nadat een bepaalde tijd is verstreken.
* De **READY-**toestand geeft aan dat het proces ready-to-run is: gereed om te worden uitgevoerd. Een proces in deze toestand is *idle* (ruststand: toestand waarin een systeem wacht, omdat er niets te doen valt), maar kan worden uitgevoerd als het de besturing over de CPU krijgt. In systemen met één processor kunnen veel READY processen staan te wachten, er kan er maar één tegelijk worden uitgevoerd. Deze toestand is niet van toepassing op een proces dat op I/O wacht.
* De toestand **RUNNING** geeft aan dat het proces wordt uitgevoerd en op dat moment onder de besturing van de CPU staat. In een single-processorsysteem is er maar één proces in deze toestand. Bevat het systeem echter meerdere CPU's, dan kunnen verscheidene processen gelijktijdig worden uitgevoerd.
* De toestand **WAIT** geeft aan dat het proces op iets staat te wachten. Het kan bijvoorbeeld een I/O request hebben geproduceerd. Het kan niet worden uitgevoerd omdat het moet wachten tot de I/O-actie klaar is. De reden voor de wachttoestand is dat I/O-snelheden laag zijn als we ze vergelijken met de CPU-snelheden. Een proces in de WAIT-toestand doet geen pogingen de besturing over de CPU te krijgen.
* De toestand **SUSPENDED** (opgeschort) is in één opzicht vergelijkbaar met de WAIT-toestand: het proces kan niet naar de CPU meedingen. Het verschilt echter in die zin, dat processen in een wait-toestand nog wel om sommige resources mogen vragen. Het proces dat een I/O aanvraag gedaan heeft, staat in feite te wachten op een I/O-processor of een I/O-controller. Het proces is idle ten opzichte van de CPU, maar het vordert nog steeds.
Een proces in de SUSPENDED-toestand kan geen enkele resource-aanvraag doen. Het proces wordt tijdelijk compleet stilgelegd. Processen kunnen bijvoorbeeld worden opgeschort wanneer het systeem te veel processen bevat.
De intensieve strijd om resources kan de voortgang van alle processen vertragen. Een van de oplossingen daarvoor is het opschorten van een paar processen, zodat het systeem de rest efficiënter kan afhandelen. Wanneer de zware competitie om resources vermindert, kan het systeem de opgeschorte processen weer activeren.
* Een proces in de toestand **COMPLETE**, tenslotte, is volledig afgewerkt.

Een toestandsovergang is een wijziging in de toestand van een proces. Bepaalde events kunnen een dergelijke wijziging vereisen. De toestandsbeschrijvingen geven u waarschijnlijk wel een indicatie wat sommige van deze events kunnen zijn, maar toch gaan we alle toestandsovergangen beschrijven**. De opdrachten die nodig zijn om alle programma's aan te bieden, duiden we aan met Job Control Language (JCL).**

In het procestoestand-diagram zijn volgende toestandovergangen:

1.De overgang van **niet-aangeboden naar HOLD** vindt plaats wanneer een gebruiker een programma aanbiedt met instructies de uitvoering van dat programma tot later uit te stellen. De JCL kan een start- of vertragingstijd bevatten. Het programma blijft in de HOLD-status tot het juiste ogenblik is aangebroken.
2.De **overgang van HOLD of niet-aangeboden naar READY** vindt plaats wanneer een gebruiker een programma aanbiedt dat van plan is direct een resource-aanvraag te doen. Deze overgang treedt ook op bij de start van een programma dat eerder op basis van vertraging is aangeboden.
3.De overgang van **READY naar RUNNING** vindt plaats wanneer het besturingssysteem de besturing van de CPU aan een bepaald proces overdraagt. De basis waarop de CPU de processen selecteert, is belangrijk en daarom het onderwerp van een volgende paragraaf (strategieën voor scheduling).
4.De overgang van **RUNNING naar READY** vindt plaats wanneer het besturingssysteem de besturing over de CPU terugkrijgt van een proces dat, indien toegestaan, nog kan runnen. Sommige systemen bepalen bijvoorbeeld hoeveel tijd een proces zonder onderbreking mag runnen (het quantum). Als het quantum is verbruikt, neemt het besturingssysteem de besturing over de CPU weer terug. Het zet het proces in de READY-toestand en geeft de CPU aan het volgende proces in de rij. Op deze manier kan geen enkel proces de CPU monopoliseren.

5.De overgang van **RUNNING naar WAIT** vindt plaats wanneer het proces iets aanvraagt waarop het moet wachten. We hebben bijvoorbeeld al gezien dat een proces dat om I/O verzoekt, moet wachten tot de I/O-actie is afgelopen.
6.De overgang van **WAIT naar READY** vindt plaats wanneer een wachtend proces krijgt wat het nodig heeft. Zo kan bijvoorbeeld het besturingssysteem een signaal ontvangen dat een I/O-actie is afgerond. Daarna zet het het wachtende proces in de READY-toestand terug.
7.De overgang van **RUNNING naar COMPLETE** vindt plaats wanneer het proces afgelopen is. Dit betekent dat de noodzakelijke werkzaamheden zijn afgerond of dat er bij het proces iets fout is gegaan en het wordt afgebroken. In beide gevallen wordt het proces beëindigd, voor zover dat het besturingssysteem betreft.
8.De overgang van **READY naar SUSPENDED** vindt plaats wanneer er teveel READY processen zijn om nog adequaat service te verlenen. Elk computersysteem heeft beperkte resources en als het aantal processen dat toegang tot deze resources vraagt, zonder beperking kan groeien, raakt het systeem verzadigd als een overvolle snelweg, zodat er filevorming optreedt en elk proces maar een gebrekkige service krijgt. Een manier om dit te vermijden is het aantal READY processen te beperken. Een andere manier is het verplaatsen van processen van de READY-toestand naar de SUSPENDED-toestand wanneer de responstijden slecht worden. Welke processen hiervoor in aanmerking komen, is natuurlijk een beslissing die het besturingssysteem moet nemen.
9.De overgang van **SUSPENDED naar READY** vindt plaats wanneer het besturingssysteem besluit dat een opgeschort proces weer naar resources kan meedingen. Vermoedelijk is de belasting tot normale niveaus teruggebracht en is het niet langer nodig om processen in de suspended-toestand te brengen.



Het besturingssysteem moet de informatie over het aantal processen op een systematische manier bijhouden. **Het besturingssysteem houdt een lijst bij van Process Control Blocks (PCB's).** In principe is er **één PCB voor elk proces**. Wanneer een proces wordt geïnitialiseerd, creëert het besturingssysteem een PCB voor dit proces en houdt een lijst van PCB's bij. Wordt een proces beëindigd, dan verwijdert het systeem het PCB uit de lijst.

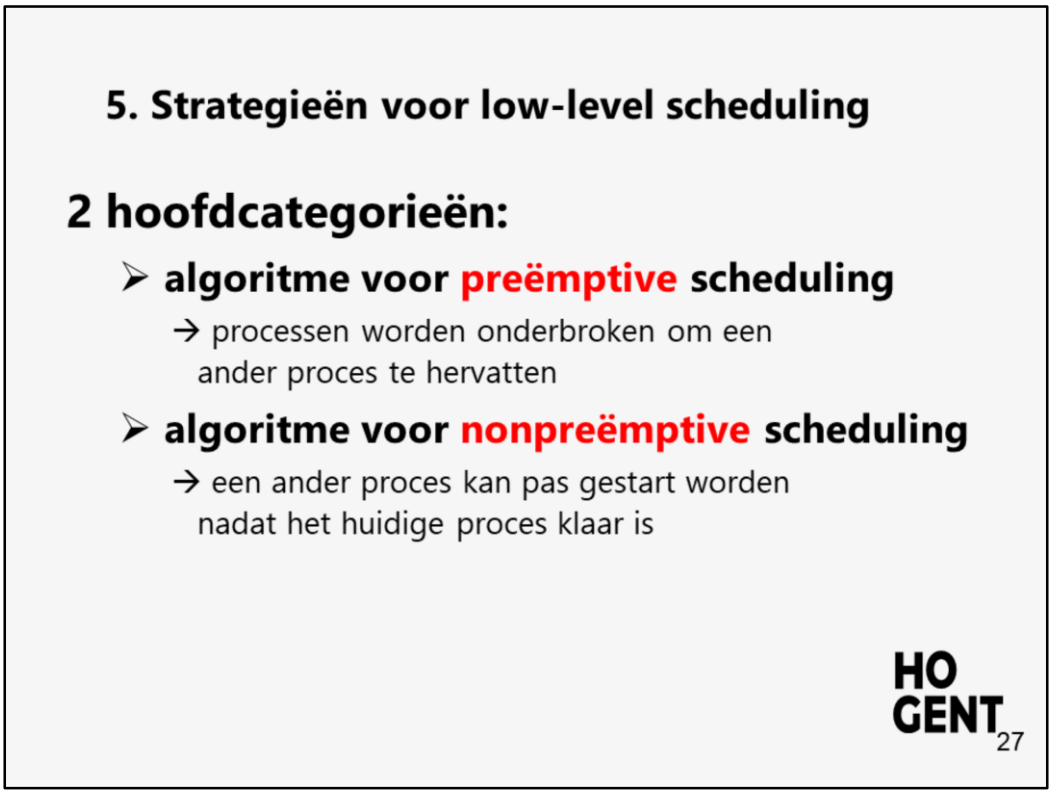
Een PCB bevat alles wat het besturingssysteem over het proces zou moeten weten:
Identificatienummer van het proces (proces-ID).
Procestoestand. Als het proces van toestand verandert, werkt het besturingssysteem het PCB van dit proces bij.
Maximale looptijd en de actuele looptijd. Gedurende de uitvoering van een proces gaat het systeem na hoeveel CPU-tijd het gebruikt. Bovendien mag de actuele looptijd de maximaal toelaatbare looptijd niet overschrijden. De maximale looptijd is een door het systeem gedefinieerde parameter en wordt bij de initialisatie van het PCB opgeslagen.
Huidige resources en limieten. Hieronder vallen onder andere het aantal printer-pagina's, de hoeveelheid geheugen en de hoeveelheid schijfruimte.
Procesprioriteit. Het systeem kan een proces schedulen of het op basis van zijn prioriteit toegang tot resources geven. Sommige processen (bijvoorbeeld routines van het besturingssysteem) hebben een hoge prioriteit.
Opslaggebieden. Als een proces tijdelijk wordt stopgezet, moet het systeem bepaalde registerwaarden bewaren. Het systeem kan deze waarden dan later herstellen, zodat het proces de uitvoering op de zelfde plek kan hervatten als waar het stopte.
Locatie van de code of de segmenttabel van een proces. Wanneer het besturingssysteem een proces laat runnen, moet het weten waar de code of de segmenttabel van dat proces zich bevindt.

Scheduling vindt plaats op hoog, middel en laag niveau: high-level, intermediate level en low-level scheduling.

High-level scheduling (job scheduling) stelt vast welke programma's of jobs toegang tot het systeem krijgen. Besturingssysteem-routines die de high-level scheduling verzorgen, controleren de Job Control Language (JCL) van een job. High-level scheduling regelt de toestandsovergangen 1,2 en 7 in het procestoestand-diagram.
High-level schedulers controleren een checklist voordat ze een job toestemming geven het systeem binnen te komen. Ze reageren eenvoudig op events en beschermen de integriteit en de veiligheid van het systeem. Er is relatief zelden behoefte aan high-level schedulers.
Scheduling op middelniveau (intermediate level) bepaalt welke processen in feite kunnen meedingen naar de CPU. Processen die I/O requests hebben geproduceerd kunnen daarbij niet meedoen. Soms schort het systeem een proces op wanneer de vraag ongewoon hoog is. Intermediate-level scheduling houdt zich voornamelijk bezig met de toestandovergangen 5, 6, 8 en 9 in het procestoestand-diagram.
Net als high-level schedulers zijn ook de intermediate-level schedulers event-gestuurd. Events als een I/O-aanvraag of I/O-einde bepalen meestal welke toestandovergang plaatsvindt.
Low-level scheduling is verantwoordelijk voor de toestandovergangen 3 en 4 in het procestoestand-diagram.

In vele opzichten is scheduling op laag niveau het moeilijkst te implementeren. In tegenstelling tot andere niveaus is deze niet hoofdzakelijk event-gestuurd. Vaak zijn er veel processen die om CPU-tijd vragen en low-level scheduling moet vaststellen welk proces toegang krijgt.

In time sharing-systemen, waar processen de CPU om de beurt gebruiken, vindt low-level scheduling veelvuldig plaats. Is er veel I/O-activiteit, dan kan het elke paar milliseconden wel nodig zijn. Algoritmen voor low-level scheduling moeten proberen de CPU en de andere resources efficiënt toe te wijzen. Ze moeten een rechtvaardige respons naar de gebruikers garanderen.



Er zijn twee hoofdcategorieën algoritmen voor low-level scheduling: algoritmen voor preemptive (preventieve) scheduling en voor nonpreemptive scheduling.

Bij **nonpreemptive scheduling** houdt een proces dat de besturing over de CPU krijgt, deze besturing tot het proces is afgelopen. Het besturingssysteem neemt het proces de besturing niet af. Nonpreemptive scheduling heeft het voordeel dat het eenvoudig is. Het besturingssysteem geeft de besturing van de CPU alleen aan een proces wanneer een ander proces afgelopen is. Daarom is scheduling op laag niveau zelden nodig bij nonpreemptive scheduling. Nonpreemptive scheduling heeft het nadeel dat het niet op andere processen reageert. Het proces dat de besturing over de CPU heeft, krijgt volledige service, maar alle andere processen moeten wachten.

Bij **preemptive scheduling** kan een proces niet oneindig lang de besturing over de CPU houden. Na een bepaalde periode kan het besturingssysteem besluiten de CPU van dit proces weg te halen. De processen in de READY-toestand moeten de CPU om beurten gebruiken. Om kort te zijn: het besturingssysteem beslist hierover.

In het algemeen kan het besturingssysteem een running proces de besturing van de CPU ontnemen om verschillende redenen, bijvoorbeeld:
het proces is beëindigd; het proces genereert een request waarop het moet wachten; de uitvoering van het proces heeft al een hele tijd geduurd. De maximale tijd die het besturingssysteem een proces zonder onderbreking laat runnen is een parameter van het besturingssysteem en wordt quantum genoemd. Als het quantum wordt bereikt, slaat het besturingssysteem de context van het proces op (statusword, de registers en de programmateller), en geeft het de CPU gedurende een bepaalde tijd aan een ander proces.

5. Strategieën voor low-level scheduling

Criteria voor scheduler:

- CPU-gebruik
- througput (doorvoersnelheid)
- wachttijd
- responstijd

GENT 28

Vereisten voor de scheduler: De verschillende algoritmen hebben verschillende eigenschappen en bij het kiezen van het algoritme voor de scheduler zal het karakter van het besturingssysteem bepalen welk algoritme gekozen wordt. Sommige systemen (bijvoorbeeld realtimebesturingssystemen) stellen strikte eisen aan het algoritme.

Een aantal criteria voor de keuze van de scheduler zijn:
→ **CPU-gebruik**: de mate waarin de processor gebruikt wordt door de verschillende processen,
→ **Debiet:** het aantal processen dat zijn werk voltooit in een bepaalde tijd, Turnaround-tijd: de tijd die nodig is om een proces te voltooien.
→ **Wachttijd:** de som van alle tijd die een proces doorbrengt in een wacht rij.
→ **Responstijd:** de tijd die nodig is om een reactie te krijgen. Het is de tijd tussen het indienen van een aanvraag en het krijgen van een antwoord.

5. Strategieën voor low-level scheduling (2)

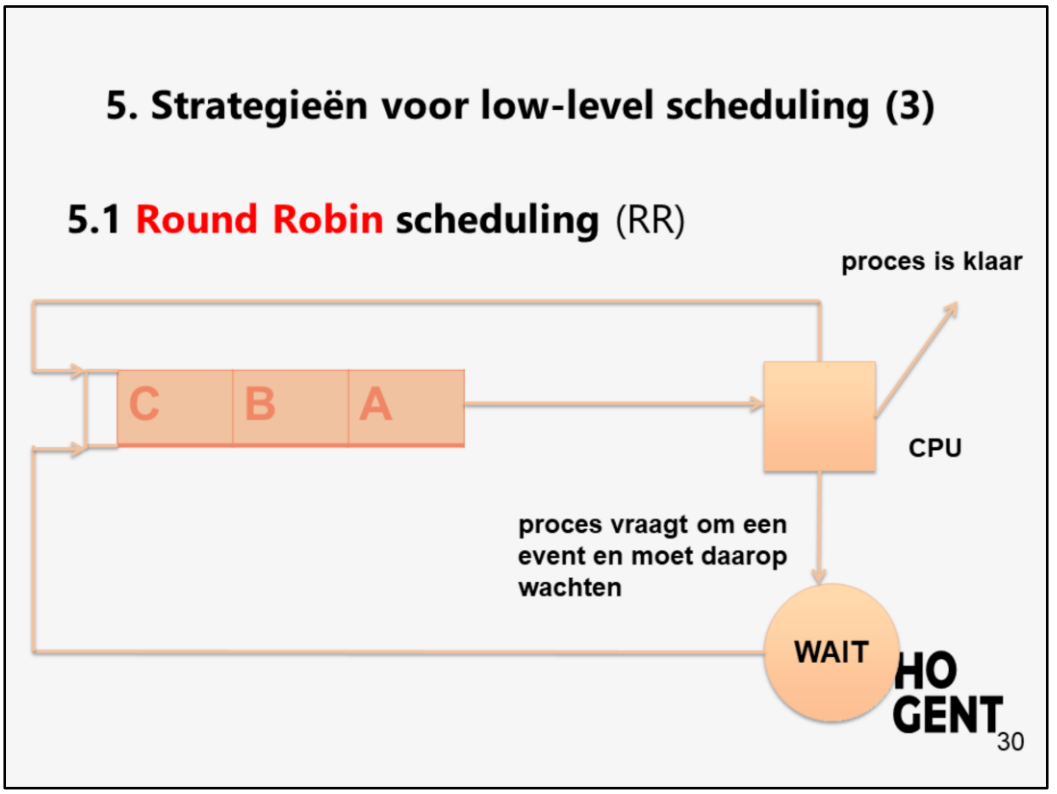
5.1 Round Robin scheduling (RR)

→Hier wordt gebruik gemaakt van **een vaste tijdswaarde of tijdkwantum**, wanneer dit overschreden wordt, zal de scheduler het proces onderbreken en een volgend proces inladen.

HO GENT 29

Round-Robin-**scheduling (RR)** → In dit scheduling algoritme wordt gebruikgemaakt van een vaste tijdswaarde, ook wel tijdkwantum genoemd. Wanneer dit tijdkwantum overschreden wordt, zal de scheduler het proces onderbreken en een volgend proces inladen. Een moeilijkheid is het bepalen van de grootte van het tijdkwantum. Een te groot tijdkwantum zal er voor zorgen dat we een FCFS[1] karakter krijgen en een te klein tijdkwantum zal voor een overhead aan context switches zorgen. Een context switch is het wisselen van processen. Wanneer de scheduler een proces onderbreekt zal hij de huidige status van het proces bewaren. Bij een te klein tijdskwantum zal de processor meer bezig zijn met het verwerken van context switches dan met het effectief uitvoeren van processen.

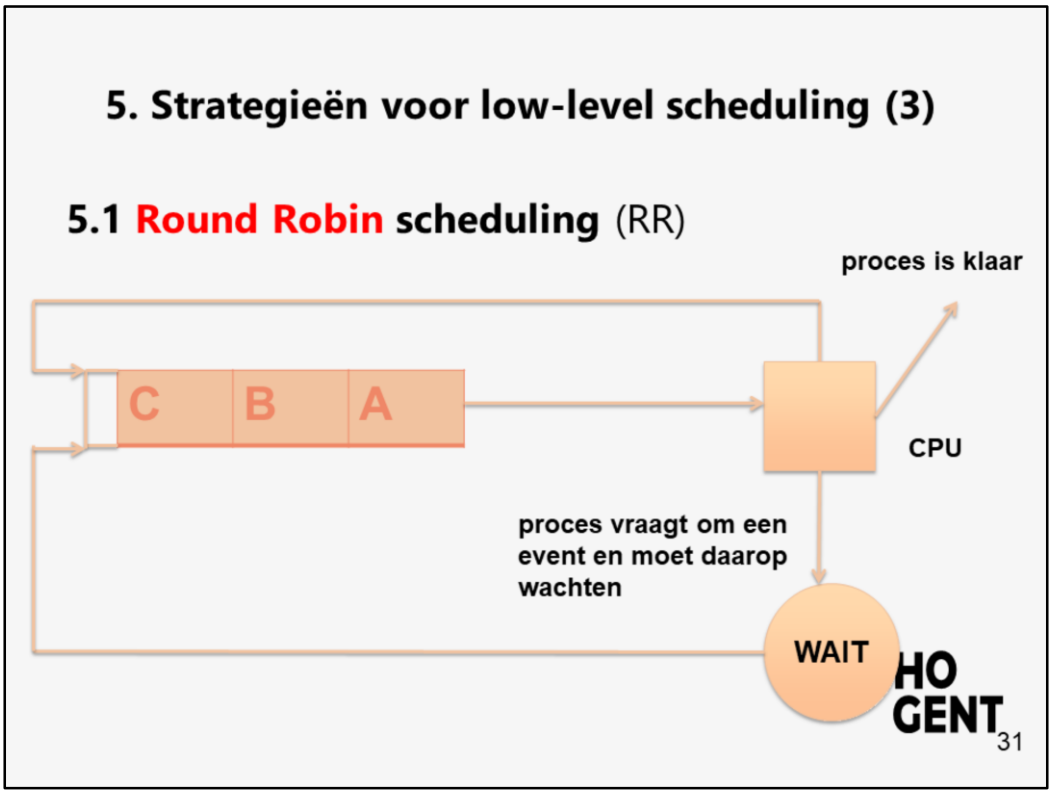
[1]FCFS: First Come First Served



De preëmptive Round Robin methode →Elk proces in de READY-toestand heeft een entry in de queue (gewoonlijk het PCB van het proces). Wanneer de CPU beschikbaar komt, geeft het besturingssysteem de besturing aan het proces waarvan het PCB aan het begin van de wachtrij staat, zodat het proces met de uitvoering begint. Het proces wordt verder uitgevoerd totdat één van de drie eerder aangegeven events optreedt. Op dat punt haalt het besturingssysteem de besturing van de CPU bij het proces weg. Als het proces afgelopen is, verwijdert de high-level scheduler het uit het systeem en verwijdert zijn PCB. Meestal creëert het besturingssysteem een overzicht van de gebruikte systeemtijd en resources, en brengt dit gebruik in rekening. Daarna bestaat het proces niet langer in het denken van het besturingssysteem.

Als het proces een request voor zoiets als een I/O heeft gegenereerd, zet het besturingssysteem het proces in de WAIT-toestand. Voor PCB's van processen die staan te wachten is er een aparte lijst. Het proces blijft in deze toestand tot het krijgt wat het nodig heeft. Heeft het een aanvraag gegenereerd, dan wacht het op een I/O-processor die de CPU een seintje geeft dat de I/O-actie is afgelopen. De intermediate-level scheduler kan het proces dan uit de blocked-toestand halen door het PCB van de wachtlijst te verwijderen. Vervolgens wordt het PCB aan het einde van de queue geplaatst. Na enige tijd krijgt het proces weer een kans de CPU te gebruiken.

Als het proces niet stopt en evenmin een aanvraag genereert waarop het moet wachten, neemt het besturingssysteem het heft in handen. Het proces mag de CPU niet langer besturen dan een kwantum. Als een kwantum is verbruikt en het proces nog steeds wordt uitgevoerd, stopt het besturingssysteem het proces en zet het PCB aan het einde van de queue. Dan geeft het de CPU aan het proces dat vooraan in de queue staat. Het voortijdig onderbroken (preëmpted) proces moet wachten tot alle andere processen in de wachtrij een gelegenheid hebben gehad om de CPU te gebruiken. Preëmption voorkomt dat processen de CPU monopoliseren. Geen enkel proces kan langer dan de kwantumtijd worden uitgevoerd zonder onderbroken te worden. Hierdoor krijgt elk proces in de READY-toestand gelegenheid gedurende een bepaalde tijd de CPU te gebruiken.



Round Robin scheduling wordt het meest gebruikt in systemen met veel, achter terminal werkende, interactieve gebruikers.

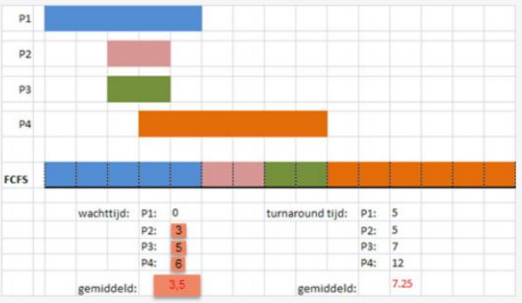
Round Robin scheduling vereist echter enige overhead. Het besturingssysteem moet de activiteiten van elk proces van heel dichtbij volgen. Een ingebouwde timer moet elk proces onderbreken dat langer dan een quantum van de CPU gebruik probeert te maken. Wanneer een dergelijke interrupt optreedt, voert het besturingssysteem een rescheduling uit en geeft de besturing van de CPU aan een ander proces. Hierdoor is het mogelijk dat het besturingssysteem vrij veel moet ingrijpen. Aangezien routines van het besturingssysteem meestal dezelfde CPU gebruiken als de processen die CPU-tijd willen hebben, vermindert dat de totale hoeveelheid tijd die voor processen beschikbaar is.

Daarnaast is de keuze van een quantumwaarde kritisch. Deze is bij Round Robin scheduling een sleutelwaarde en moet zorgvuldig gekozen worden.

5. Strategieën voor low-level scheduling (4)

5.2 First-in-First-Out scheduling (FIFO) of First-come-first-served-scheduling (FCFS)

→Wanneer een proces als eerste de CPU vraagt zal hij die ook krijgen, waarbij de andere processen die erna komen zullen moeten wachten.



HO GENT 32

First-come-first-served-scheduling (FCFS) → Dit is het eenvoudigste algoritme. Wanneer een proces als eerste om de cpu vraagt dan zal hij die ook krijgen. Processen die erna komen moeten wachten. Deze vorm van scheduling kan geïmplementeerd worden door een First-in-first-out (FIFO) model. Wanneer een proces lang zal duren, zullen korte processen die erna komen lang moeten wachten.

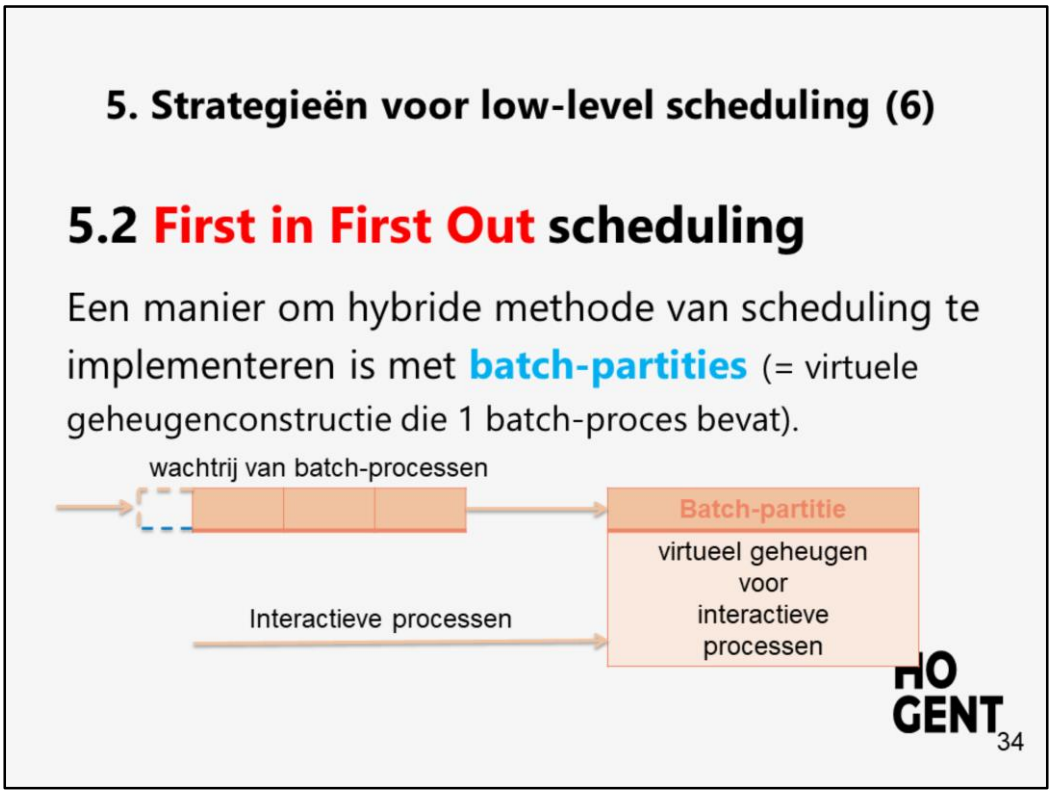
Deze nonpreemptive-methode is heel eenvoudig: geef de besturing van de CPU aan het proces dat zich het langst in het systeem bevindt. Het proces dat het eerst aankwam mag de CPU gebruiken.

Wanneer een proces is geïnitialiseerd, plaatst het systeem het PCB van het proces aan het einde van een queue. Wanneer de CPU beschikbaar komt, geeft het besturingssysteem de besturing van de CPU aan het proces dat vooraan in de queue staat. Het proces houdt de controle over de CPU totdat het is afgelopen en daarna geeft het besturingssysteem de CPU aan een ander proces.

Het belangrijkste voordeel van deze methode is zijn eenvoud. Dit is iets wat nooit mag worden onderschat. Het besturingssysteem besluit alleen tot rescheduling als dat absoluut noodzakelijk is. De overhead is daardoor klein.

In vele systemen kan FIFO onrealistisch zijn (bijvoorbeeld in een hoog-interactief systeem met veel I/O-activiteit), toch zijn er gevallen waarin FIFO gewenst is (bijvoorbeeld een systeem waarin de meeste processen zwaar rekenwerk verrichten, maar erg weinig I/O-aanvragen produceren).

Een FIFO-scheduler kan ook deel uitmaken van een ingewikkelder methode: bijvoorbeeld bij systemen die zowel batch-gebruikers als interactieve gebruikers hebben. Interactieve gebruikers laten korte programma's draaien, hebben weinig informatie nodig of ontwikkelen nieuwe applicatie en hebben een snelle respons nodig. Interactieve gebruikers op terminals verwachten bijna onmiddellijk respons van het systeem. Korte onderbrekingen in de uitvoering worden direct opgemerkt. Een batch-gebruiker daarentegen biedt zijn of haar programma op een bepaalde tijd aan, samen met de JCL-opdrachten die nodig zijn om het te laten draaien. Nadat het is aangeboden, is de gebruiker echter vrij om iets anders te gaan doen. In tegenstelling tot de interactieve gebruiker hoeft hij of zij tijdens de verwerking van het proces niet aanwezig te zijn. De programma's van batch-gebruikers zijn meestal langer of willen veel uitvoer produceren. In tegenstelling tot interactieve gebruikers merken batch-gebruikers niets van korte onderbrekingen. Het besturingssysteem kan dit gebruiken om aan beide gebruikers een goede service te bieden.



Wanneer batch-processen en running processen worden gemengd, is een Round Robin schedule van alle processen niet gewenst, met name als er veel batch-gebruikers zijn. Deze hebben geen snelle respons nodig. Waarom zouden we dan niet wat CPU-tijd van batch-processen stelen en die aan de interactieve processen ter beschikking stellen. De batch-gebruikers merken het verschil niet eens en de interactieve gebruikers zullen de verbeterde respons waarderen.

Een manier om deze hybride methode van scheduling te implementeren is met batch-partities. Een batch-partitie is een virtuele geheugenconstructie die één batch-proces bevat. Een virtueel geheugen is een geheugenbeeld dat het besturingssysteem aan de gebruiker verschaft. Er kunnen verscheidene batch-partities zijn, maar in dit voorbeeld gaan we uit van één.

Als de processen het systeem binnenkomen, maakt het systeem verschil tussen de batch-processen en de interactieve processen. Alle interactieve processen komen direct in de READY-toestand en dingen mee naar de CPU-tijd. Het systeem zet de batch-processen echter in een wachtrij voor de batch-partitie. Aangezien de partitie slechts één proces kan bevatten, moeten alle andere wachten.

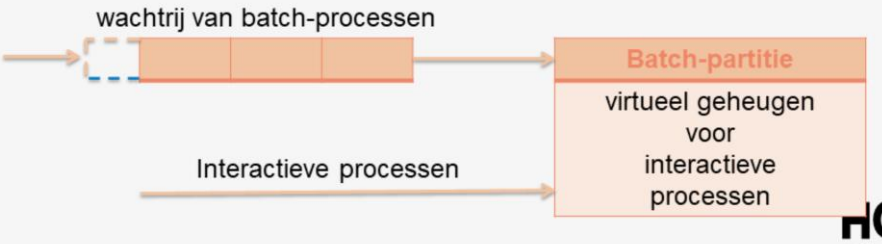
Als de partitie leeg is, wordt het proces aan het begin van de queue in de partitie geplaatst. Het proces komt in de READY-toestand en dingt samen met de interactieve processen mee naar de CPU-tijd. Is de partitie vol, dan moet het proces dat vooraan in de queue staat wachten. Het resultaat is een quota-systeem waarbij slechts één batch-proces tegelijk de gelegenheid krijgt naar de CPU mee te dingen.

Hierin is de low-level scheduler Round Robin, maar er is nooit meer dan één batch-proces READY, en de intermediate-level FIFO-scheduler bepaalt welke processen in de READY-toestand komen.

5. Strategieën voor low-level scheduling (6)

5.2 First in First Out scheduling

Een manier om hybride methode van scheduling te implementeren is met **batch-partities** (= virtuele geheugenconstructie die 1 batch-proces bevat).



HO GENT 35

Round Robin en FIFO zijn twee veel voorkomende methoden bij de scheduling op laag- en middelniveau, maar het zijn geenszins de enige methoden.

Round Robin is het meest geschikt wanneer gebruikers een snelle respons willen.

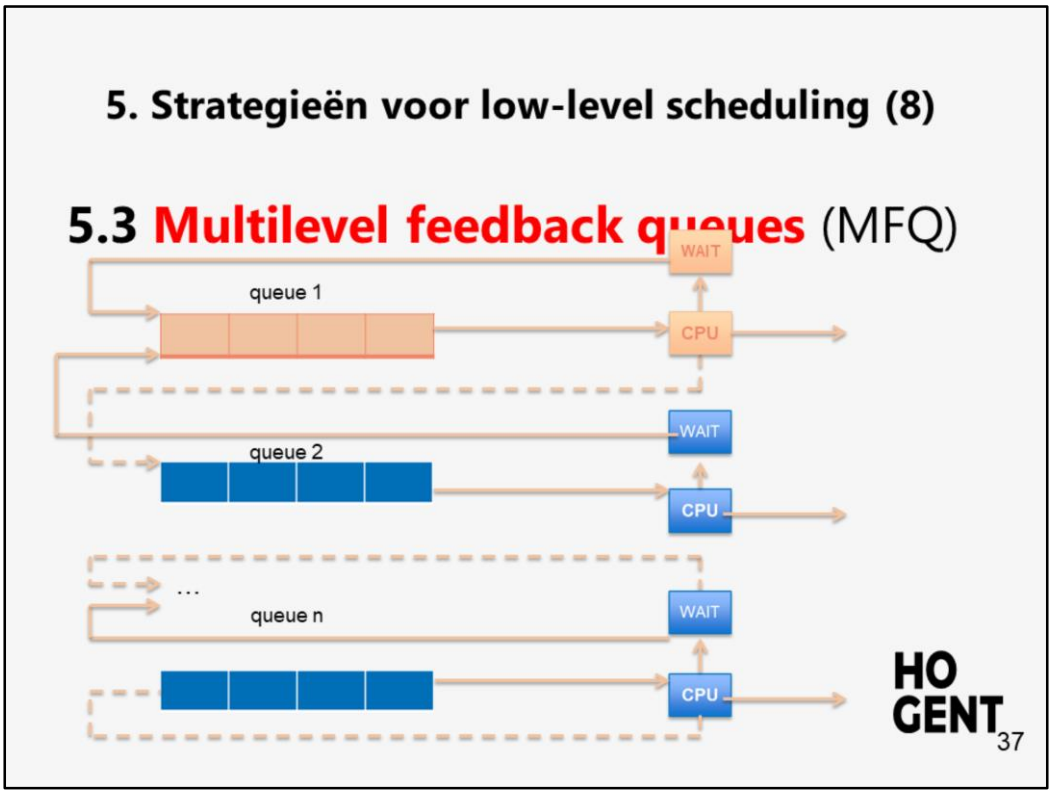
Round Robin past heel goed in systemen die batch-processen verwerken die veel I/O requests genereren. In dit geval willen we dat de processen hun eigen aanvragen zo spoedig mogelijk genereren. Dit houdt de I/O-processors actief en verkleint het aantal processen in de READY-toestand. Aangezien Round Robin garandeert dat alle READY-processen een kans krijgen om te worden uitgevoerd, vergroot deze de kansen dat er I/O requests worden gegenereerd.

FIFO daarentegen is het meest geschikt voor omgevingen waarin zwaar rekenwerk moet worden verricht of als onderdeel van een ingewikkelder scheduling-benadering. Er is niets te winnen door FIFO in interactieve omgevingen te gebruiken of wanneer er veel I/O-activiteit bestaat. Omgekeerd is er evenmin winst als we Round Robin in een rekenomgeving zouden toepassen.

Computeromgevingen zijn dynamisch en het soort processen dat het systeem binnenkomt kan enorm variëren. Als we óf Round Robin óf FIFO gebruiken, zullen er momenten zijn waarop de scheduler zijn werk niet zo best doet.

Er bestaat een manier waarop de scheduling-methode op Round Robin kan lijken als er veel I/O-activiteit is en op FIFO kan lijken wanneer er weinig of geen I/O-activiteit is: dit wordt **Multilevel Feedback Queues (MFQ)** genoemd. Bij deze methode is de scheduling afhankelijk van de activiteiten die op een bepaald moment plaatsvinden. De beste scheduling-methode is afhankelijk van de soorten processen in de READY-toestand en het MFQ-systeem is gevoelig voor wijzigingen in die activiteiten (adaptieve methode).

Wanneer er veel I/O-activiteit plaatsvindt, lijkt MFQ op een Round Robin scheduler. Is er daarentegen maar weinig of geen I/O-activiteit, dan lijkt MFQ op een FIFO-scheduler. MFQ is nooit compleet FIFO, omdat het dan niet op veranderingen zou kunnen reageren in de tijd dat een bepaald proces de besturing over de CPU heeft. Het voert echter minder reschedules uit door elk proces de gelegenheid te geven gedurende langere tijd de CPU te bestuderen.



Zoals de naam suggereert, bestaat MFQ uit vele queues. Ze zijn genummerd van 1 tot n en elke wachtrij heeft een eigen prioriteit (q_1 heeft de hoogste prioriteit en q_n heeft de laagste prioriteit). Het PCB voor een proces in de READY-toestand is in één van de queues geplaatst. De prioriteit van het PCB wordt bepaald door de queue waarin het zich bevindt.

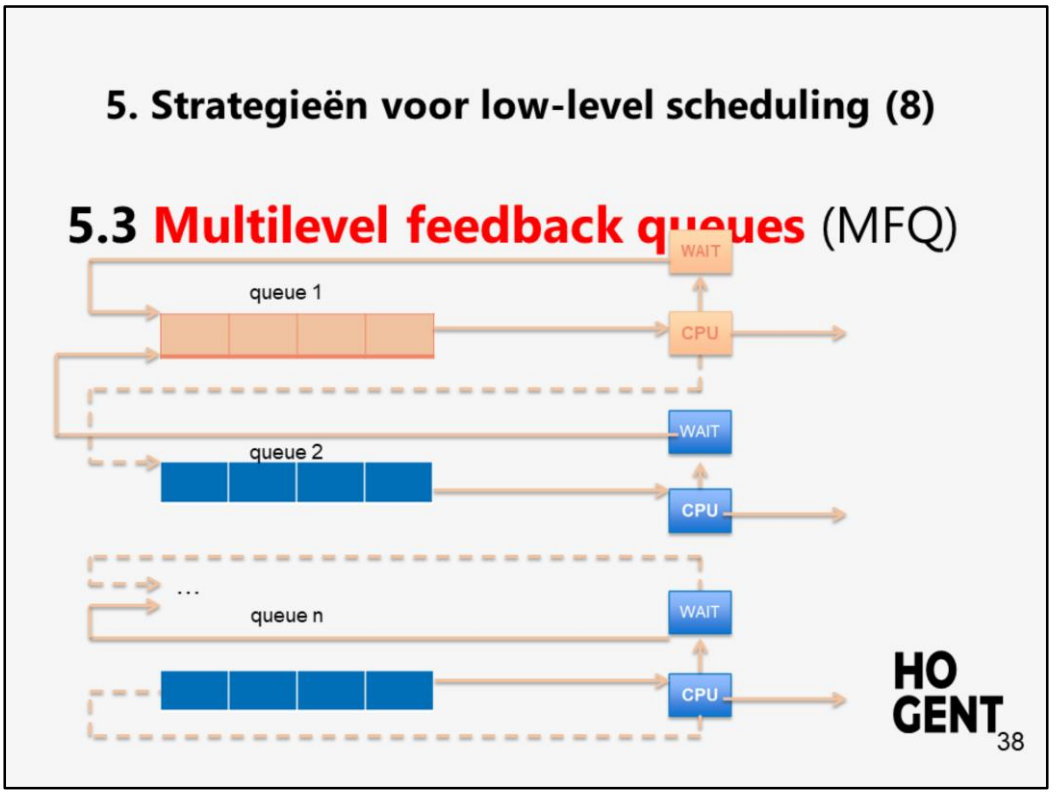
De low-level scheduler werkt altijd op basis van prioriteit. Een proces met lage prioriteit krijgt alleen de besturing van de CPU als er geen proces met een hogere prioriteit bestaat. Heeft een aantal processen dezelfde prioriteit, en is er geen met een hogere, dan kiest de scheduler het proces aan het begin van de queue (FIFO).

Telkens wanneer een nieuw proces wordt geïnitialiseerd, zet het systeem het in de READY-toestand en wordt het PCB van het proces in de eerste queue geplaatst. Elk nieuw proces heeft aan het begin dus de hoogste prioriteit. Is de CPU beschikbaar, dan zoekt de low-level scheduler de queue met de hoogste prioriteit die niet leeg is. Dan geeft de low-level scheduler de besturing van de CPU aan het proces dat vooraan in die queue staat.

Het proces krijgt de besturing van de CPU tot één van de volgende dingen gebeurt:
- het proces wordt beëindigd (elegant of door het af te breken);
- het proces vraagt iets aan waarop het moet wachten (bijvoorbeeld een I/O request);
- het quantum is verbruikt.

Het quantum is afhankelijk van de queue waaruit het proces is gekozen. Het is kenmerkend dat elke queue een ander quantum heeft. Queue 1 heeft het kleinste quantum en queue n heeft het grootste quantum.

Als het proces afgelopen is, verlaat het het systeem en speelt het verder geen rol.



Als het proces iets aanvraagt waarop het moet wachten, zet de intermediate-level scheduler het in de WAIT-toestand. Wanneer het proces krijgt wat het nodig heeft, keert het terug naar de READY-toestand. Het besturingssysteem plaatst het PCB aan het einde van de queue met net iets hogere prioriteit dan de queue waarin het heeft gestaan. Het besturingssysteem verhoogt de prioriteit van het proces.

Stel dat het proces geen enkele aanvraag doet en niet stopt. Dan haalt het besturingssysteem de besturing van de CPU bij het proces weg nadat het quantum is verbruikt. De quantumwaarde is afhankelijk van de queue die het PCB van het proces bevatte. In dit geval blijft het proces in de READY-toestand en het besturingssysteem plaatst het PCB aan het einde van de queue met net een iets lagere prioriteit dan de queue waarin het heeft gestaan.

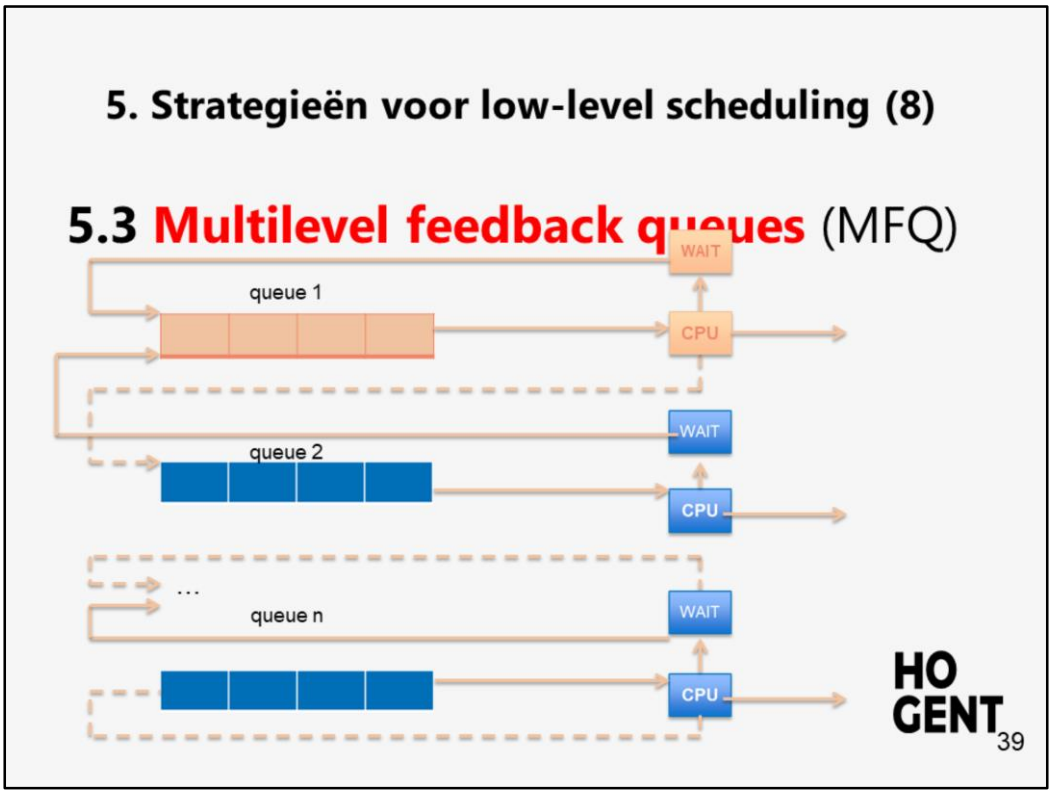
Hoe zorgt het besturingssysteem ervoor dat de low-level scheduling zich aanpast aan het type processen die READY zijn?

Al de processen ontvangen in eerste instantie de hoogste prioriteit.

Als de processen interactief zijn of I/O-gebonden, ontvangt het besturingssysteem geregeld I/O-aanvragen. Stel dat het quantum iets langer is dan de gemiddelde tijd tussen I/O requests. Hierdoor genereren de meeste processen I/O-requests voordat het quantum is verbruikt.

Wanneer een I/O-request is afgelopen, komt het proces uiteindelijk opnieuw in de READY-toestand, met een hogere of dezelfde prioriteit. Daarom heeft elk I/O-gebonden proces de neiging een hoge prioriteit te behouden en meer I/O-requests te genereren.

Als het proces enige tijd wordt uitgevoerd zonder een I/O-aanvraag, raakt het quantum op. In dit geval blijft het proces in de READY-toestand, maar met een lagere prioriteit (tenzij het de laagste prioriteit heeft). Is het proces echter nog I/O-gebonden, dan zal het spoedig meer I/O-aanvragen genereren en weer hogere prioriteit krijgen.



Stel dat de binnenkomende processen echt rekenintensief zijn. Deze zullen weinig I/O requests genereren, maar de meeste tijd spenderen aan het uitvoeren van code. Wanneer ze de besturing van de CPU ook krijgen, meestal zal het quantum worden verbruikt. Het gevolg is dat ze met een lagere prioriteit naar de READY-toestand terugkeren (tenzij ze al de laagste prioriteit hebben).

Als alle READY-processen rekenintensief zijn, krijgen ze uiteindelijk allemaal de laagste prioriteit. De low-level scheduler heeft dus vooral te maken met processen met een lage prioriteit. Het wordt een Round Robin scheduler met een groter quantum dan voor I/O-gebonden processen wordt gebruikt. De grotere quanta zijn gerechtvaardigd omdat er met geregelde rescheduling van rekenintensieve processen niets wordt gewonnen. Hoewel de scheduler nog steeds Round Robin is, zorgen de grotere quanta voor een werkwijze die meer op FIFO lijkt.

Laten we vervolgens aannemen dat het gaat om een mengsel van I/O-gebonden en CPU-gebonden processen. De I/O-gebonden processen behouden een hoge prioriteit, terwijl de prioriteit van de CPU-gebonden processen afneemt. De scheduler begunstigt de I/O-gebonden processen boven de CPU-gebonden processen. Door dit te doen, blijven de I/O-processors actief en neemt de graad van multiprocessing toe. CPU-gebonden processen worden alleen uitgevoerd wanneer er geen I/O-gebonden processen READY zijn.

Niet alleen is MFQ gevoelig voor verschillen tussen processen, het is ook gevoelig voor veranderingen binnen een proces. Zo kan een proces van start gaan als I/O-gebonden, maar opeens veranderen in een CPU-gebonden proces. Het omgekeerde kan ook voorkomen.

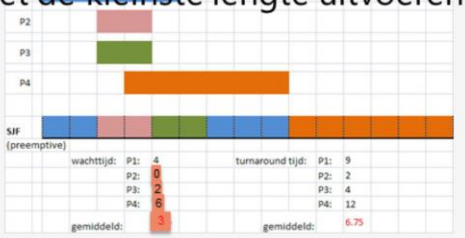
Het systeem met queues op verschillende niveaus (MFQ) is een flexibele methode die zorgt voor een automatische aanpassing aan de werkbelasting en aan veranderingen in het gedragspatroon van processen.
Als een systeem hoofdzakelijk interactieve of CPU-gebonden processen heeft, is MFQ ingewikkelder dan nodig. In een onvoorspelbare omgeving kan deze methode echter productief zijn. Dit is in feite de scheduling-methode die door VMS wordt toegepast.

5. Strategieën voor low-level scheduling (9)

5.4 **Shortest-job-first-scheduling** (SJF)

Er zijn twee strategieën die aan korte processen een hoge prioriteit geven:

- **Shortest Remaining Job Next (SRJN)** = preëmptieve versie van SJF
- **Shortest Job First (SJF)** → hier zal de scheduler het proces met de kleinste lengte uitvoeren



HO GENT

De vorige methoden zijn best geschikt voor verschillende soorten computeromgevingen. Ze doen het echter allemaal wat minder goed als er te veel processen zijn. Als bijvoorbeeld de scheduler van het type FIFO is en de queue veel processen bevat, betekent dat lange wachttijden voor de processen achter in de rij. De vertraging is met name vervelend voor korte processen die achter langere processen staan te wachten. Vergelijkbare problemen kunnen zich voordoen bij een Round Robin scheduler. Als er te veel processen zijn, heeft de scheduler meer tijd nodig om ze allemaal een beurt te geven. Het gevolg is dan aanmerkelijk langere responstijden.

Is het logisch om korte processen een hogere prioriteit te geven om ze sneller uit het systeem te krijgen? Als dat zo is, kan het systeem het aantal processen dat om resources vraagt kleiner houden.
Het welslagen of mislukken van de vorige methoden is sterk afhankelijk van het type activiteit dat de running processen nodig hebben. Daarom maakten we onderscheid tussen I/O-gebonden en CPU-gebonden processen, maar nooit tussen lange en korte processen. De bovenstaande voorbeelden suggereren dat we wel een dergelijk onderscheid misschien wel zouden moeten maken.

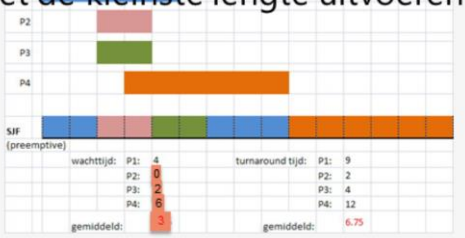
Er zijn twee strategieën die aan korte processen een hoge prioriteit geven:
- **Shortest-job-first-scheduling (SJF)** → Bij dit algoritme zal de scheduler het proces met de kleinste lengte uitvoeren. Een probleem is het voorspellen van de lengte van een proces. Dit kan gedaan worden door bijvoorbeeld het aantal instructies te tellen of het aantal in- en uitvoer aanvragen. Een voordeel ten opzichte van het FCFS-algoritme is dat de wachttijd bij SJF kleiner is. Het SJF-algoritme bestaat in twee vormen, de preemptive en de non-preemptive vorm. Een gevaar bij SJF-scheduling is dat een heel lang proces nooit uitgevoerd zal worden. Dit noemt men “verhongering” (Eng: starvation).
- **Shortest Remaining Job Next (SRJN)**
Ze zijn in die zin vergelijkbaar, dat de low-level scheduler rekening houdt met de hoeveelheid tijd die een READY-proces nodig heeft. Eenvoudig gesteld, het kiest het proces dat de minste tijd nodig heeft. Het verschil tussen de twee strategieën is dat SRJN preemptive is en SJF nonpreemptive is.

5. Strategieën voor low-level scheduling (9)

5.4 **Shortest-job-first-scheduling** (SJF)

Er zijn twee strategieën die aan korte processen een hoge prioriteit geven:

- **Shortest Remaining Job Next (SRJN)** = preëmptieve versie van SJF
- **Shortest Job First (SJF)** → hier zal de scheduler het proces met de kleinste lengte uitvoeren



HO GENT 41

Een nadeel van beide methoden is dat het besturingssysteem het moeilijk heeft om te berekenen hoeveel tijd een proces nodig heeft. Het is moeilijk (in feite meestal onmogelijk) voor een besturingssysteem om te berekenen hoeveel tijd een proces nodig heeft. De persoon die het proces heeft aangeboden, zal echter best een redelijke schatting kunnen geven (zo niet, dan zou hij of zij de job waarschijnlijk beter niet kunnen aanbieden). Dergelijke ramingen maken in feite deel uit van de vereiste JCL. De low-level scheduler kan deze schatting dus gebruiken om te beslissen welk proces de besturing over de CPU krijgt. Bij SJF neemt de low-level scheduler het READY-proces met de korst geschatte run-tijd. Heeft dat eenmaal de besturing van de CPU, dan draait het tot het is afgelopen (non preemptive).

Nonpreeptive scheduling creëert problemen als er I/O requests zijn. Maar, als er maar weinig of geen I/O wordt verwacht, is SJF uitvoerbaar. Korte processen krijgen een prima respons omdat ze de hoogste prioriteit hebben. De doorvoer-snelheid is indrukwekkend. In feite genereert SJF de hoogst mogelijke doorvoer-snelheid. Als het erom gaat een zo groot mogelijk aantal tevreden gebruikers te krijgen, is dit de beste methode.

Het is echter mogelijk dat de doorvoersnelheid een verkeerde indruk van de pro-duktiviteit van het systeem geeft. Kijk maar naar de gebruiker wiens proces iets langer loopt dan veel andere. Het systeem straft dit af. Uiteraard kan het lei-den tot indefinite postponement (onbepaald uitstel) of starvation (uithongeren); dat wil zeggen, het kan eeuwig staan wachten. Zolang kortere processen het systeem blijven binnenkomen, zullen de langere niet worden uitgevoerd.

Het kan in hoge mate onrechtvaardig zijn om een proces uren te laten wachten, terwijl een iets korter proces wel wordt ingevoerd en het systeem al heel snel weer verlaat. Dit is vooral het geval als het langere proces van uzelf is.

De preemptive versie van SJF is SRJN. Een proces dat de besturing over de CPU heeft, kan hiervan afstand doen als het om iets vraagt waarop het moet wachten. Als een nieuw proces een kortere geschatte run-tijd heeft dan het proces dat op dat moment wordt uitgevoerd, krijgt het de besturing van de CPU. Net als SJF, genereert SRJN een hoge doorvoersnelheid, maar het lijdt ook aan dezelfde nadelen. Langere processen kunnen in onaanvaardbare mate worden vertraagd.

5. Strategieën voor low-level scheduling (10)

5.5 Starvation

→Wanneer een heel lang proces nooit uitgevoerd zal worden, noemen we dit starvation.

Kunnen we iets aan starvation doen of moeten we ermee leven en het als bijproduct van een bepaalde methode beschouwen?

HO GENT 42

We hebben gezien dat de methoden SJF en SRJN starvation van langere processen kunnen veroorzaken. De MFQ-methode kan ook starvation van processen in de lagere queues tot gevolg hebben. Zolang de interactieve processen READY zijn, worden de CPU-gebonden processen genegeerd. Kunnen we iets aan starvation doen of moeten we ermee leven en het als bijproduct van een bepaalde methode beschouwen?

Eén mogelijkheid is dat we **het negeren** en hopen dat het geen ernstige problemen veroorzaakt. Dit is niet altijd een onrealistische benadering. Onder MFQ bijvoorbeeld verlaten interactieve processen de queues snel. Als gevolg van het verschil tussen de reken- en de I/O-snelheid duurt het voor deze processen veel langer om terug te keren. Daarom is het niet ongewoon te constateren dat de queues met hoge prioriteit snel leeg zijn. Natuurlijk ontstaat hierdoor voor de processen met lage prioriteit een gelegenheid om te runnen. Bij de methoden SJF en SRJN kan men vergelijkbare observaties uitvoeren. De ongelijkheid tussen de verwerkingssnelheden van de computer en de mens is hier zelfs nog groter. Er zijn waarschijnlijk niet veel gebruikers die korte processen sneller kunnen genereren dan een CPU deze kan uitvoeren. In deze gevallen is starvation zeldzaam. Zelfs wanneer het optreedt, houdt het niet lang aan. Ook al zeggen we dat events waarschijnlijk niet voorkomen, dat wil niet zeggen dat zij onmogelijk zijn. Starvation van CPU-gebonden processen onder MFQ is aannemelijker als het aantal interactieve processen toeneemt. Op dezelfde manier is starvation van lange processen onder SJF of SRJN waarschijnlijker als het aantal gebruikers toeneemt. In dergelijke gevallen is starvation een ernstiger probleem.

Als we ervoor kiezen het **niet te negeren**, wat kan het systeem er dan aan doen. Eén methode is het opschorten van een aantal READY-processen. Dit reduceert het aantal processen en vermindert starvation. Het systeem hervat de processen wanneer het besluit dat het er meer aankan.

Een ander alternatief is dat het systeem **de prioriteiten periodiek opnieuw berekent**. Het systeem kan dit doen met behulp van het veld LastTime, dat zich in het PCB van elk proces bevindt. In het begin wordt daar de tijd opgeslagen waarop het proces het systeem is binnengekomen. Telkens wanneer een proces wordt onderbroken (preempted), vervangt het systeem de waarde in het veld door het tijdstip van preemption. Onder nonpreemptive scheduling verandert het veld nooit. Wanneer het systeem de prioriteit van een proces evalueert, bekijkt het de waarde 'huidige tijd - LastTime'. Een grote waarde betekent dat het proces al lange tijd is genegeerd. Als de waarde toeneemt tot voorbij een bepaalde drempelwaarde, verhoogt het systeem de prioriteit voor dat proces. Onder MFQ betekent dit dat het PCB in een hogere queue wordt geplaatst. Onder SJF of SRJN betekent het dat het prioriteitsveld van het PCB wordt veranderd.

Bij Round Robin en FIFO kan geen starvation optreden.

5. Strategieën voor low-level scheduling (12)

	Round Robin	FIFO	MFQ	SJF	SRJN
Doorvoer-snelheid	kan laag zijn als quantum te klein is		kan laag zijn als de quanta te klein zijn	hoog	hoog
Responstijd	kortste gemiddelde responstijd, als quantum juist is gekozen	kan gebrekkig zijn, vooral als een lang proces de besturing over de CPU heeft	goed voor I/O-gebonden processen, maar kan gebrekkig zijn voor de CPU-gebonden processen	goed voor korte processen, maar kan gebrekkig zijn voor langere processen	goed voor korte processen, maar kan gebrekkig zijn voor langere processen
Overhead	laag	de laagste van alle methoden	kan hoog zijn; ingewikkelde datastructuren en routines zijn nodig om na elke reschedule de juiste queue te vinden	kan hoog zijn; vereist een routine om voor elke reschedule de kortste job te vinden	kan hoog zijn; vereist een routine om voor elke reschedule de minimale resterende tijd te vinden
CPU-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	krijgt lage prioriteit als de I/O-gebonden processen aanwezig zijn	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen
I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	krijgt hoge prioriteit om I/O-processors actief te houden	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen	geen onderscheid tussen CPU-gebonden en I/O-gebonden processen
Onbepaald uitstel	treedt niet op	treedt niet op	kan optreden bij CPU-gebonden processen	kan optreden bij processen met lange geschatte runtijden	kan optreden bij processen met lange geschatte runtijden

HO GENT 44